# Database Triggers (PostgreSQL)

## Database Trigger Overview

A trigger is an object that is automatically executed when a specified DML event occurs on a table. A trigger:

- is tied to one specific table
- is tied to one specific DML event (INSERT, UPDATE, or DELETE)
- cannot be executed manually
- cannot accept parameters
- can only fire in response to DML statements
- can fire before or after the DML operation
- can fire once per statement or once per row

## Types of Triggers

PostgreSQL provides two levels of triggers:

- Row-Level Triggers: Fires once for each row affected by the DML event.
- Statement-Level Triggers: Fires once per DML event, regardless of the number of rows affected.

## Timing of Triggers

Triggers can be specified to fire before or after the DML event alters the database:

- BEFORE Trigger: Fires before the DML event occurs. Can be used to modify or skip the DML event.
- AFTER Trigger: Fires after the DML event occurs. All changes made by the DML event are available to the trigger.
- INSTEAD OF Trigger: Fires only on Views (Not taught in this course).
- If more than one trigger in the same level/timing exists, they are executed in alphabetical order according to the trigger name.

Creating Triggers

- A trigger is implemented using a special function associated to a trigger. To create a trigger, you must define a trigger function first and then bind this trigger function to a trigger. The trigger function must be declared as a function taking no arguments and returning type trigger. The difference between a trigger function and a stored function is that a trigger function is automatically called when a trigger fires, and a stored function can be executed by a programmer.

Triggers are often used to:

- Enforce complex business rules when Check constraints are insufficient (e.g. multiple tables).
- Automate operations such as logging, archiving, or updating summary tables.
- Create audit trails by recording who changed what and when.
- Enforce referential integrity between databases (Not taught in this course).

Trigger Flow in PostgreSQL

When a DML statement executes:

- Row-Level triggers:
    - New contains the after image of a single row (Insert/Update).
    - Old contains the before image of a single row (Update/Delete).
- Statement-Level triggers:
    - New and Old transition tables may be referenced in After triggers using the Referencing keyword.
- New and Old (either record or transition table) have the same column names and data types of the table that the trigger is created on. Transition tables are temporary, read-only tables that exist only inside the trigger function and contain copies of the affected rows.

| Statement | Old | New |
|---|---|---|
| INSERT | (empty) | new row(s) |

| | | |
|---|---|---|
| UPDATE | old row(s) | new row(s) |
| DELETE | deleted row(s) | (empty) |

Simplified Trigger Syntax

PostgreSQL requires two parts:

A trigger function:

```
CREATE OR REPLACE FUNCTION function_name()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
BEGIN
    -- logic here
END;
$$;
```

A trigger:

```
CREATE TRIGGER trigger_name
{ BEFORE | AFTER } { DELETE [OR] | INSERT [OR] | UPDATE [OF column_name1 [, column_name2 ... ]] [OR] }
ON table_name
FOR EACH { ROW | STATEMENT }
EXECUTE FUNCTION function_name();
```

Dropping Triggers

```
DROP TRIGGER trigger_name ON table_name;
```

Create a trigger to display Old and New values when updating a record.

```
Create Function TRF_Display_Updated_Position ()
Returns Trigger
Language plpgsql
As $body$
Begin
	-- would normally do a Raise Notice but Raise Exception will rollback any changes
	Raise Exception 'New description % Old description %', New.PositionDescription, Old.PositionDescription;
	Return New;
End;
$body$;

Create Trigger TR_Display_Updated_Position
Before Update
On Position
For Each Row
Execute Function TRF_Display_Updated_Position ();
```

Testing

```
Select	PositionID, PositionDescription
From	Position
Where	PositionID = 6;

Update		Position
Set	PositionDescription = 'IT Support'
Where	PositionID = 6;

Select	PositionID, PositionDescription
From	Position
Where	PositionID = 6;
```

We can use a trigger to enforce a business rule (BalanceOwing cannot be less than zero) instead of using a Check constraint. The “Of BalanceOwing” added to Update has no effect on inserts but ensures that the trigger only fires when the balance owing column is updated.

```
Create Function TRF_Student_PositiveBalanceOwing()
Returns Trigger
Language plpgsql
As $body$
Begin
        If New.BalanceOwing < 0 Then
                Raise Exception 'BalanceOwing % is invalid. BalanceOwing cannot be
        less than zero.', New.BalanceOwing;
        End If;
        Return New;
END;
$body$;

Create Trigger TR_Student_PositiveBalanceOwing
Before Insert Or Update Of BalanceOwing
On Student
For Each Row
Execute Function TRF_Student_PositiveBalanceOwing();
```

Testing

```
Begin;
Select          StudentID, BalanceOwing
From            Student
Where           StudentID = 200495500;

Update                  Student
Set             BalanceOwing = -1
Where           StudentID = 200495500;
Rollback;
```

Create a trigger that enforces a rule that a Course Cost cannot be increased by more than double in one update.

```
Create Function TRF_Course_DoubleCostUpdate_Check()
Returns Trigger
Language plpgsql
As $body$
Begin
        If New.CourseCost > Old.CourseCost * 2 Then
                Raise Exception 'Course Cost change must not be more than double';
        End If;
        Return New;
End;
$body$;

Create Trigger TR_Course_DoubleCostUpdate_Check
Before Update Of CourseCost
On Course
For Each Row
Execute Function TRF_Course_DoubleCostUpdate_Check();
```

Testing

```
Select      CourseID, CourseCost
From        Course
Where       CourseID = 'COMP1017';

Update              Course
Set         CourseCost = 905
Where       CourseID = 'COMP1017';
```

We can use triggers to create a record of changes to other records. This is often called Logging and involves entering a record in a changes table when changes are made to another table. Create a table and trigger to log when changes are made to the Mark in the Registration table. If the mark does not change, do not insert a record.

```
Create Table Registration_Mark_Changes
(
	StudentID	integer		Not Null,
	CourseID	char(8)		Not Null,
	Semester	char(5)		Not Null,
	Old_Mark	decimal(5,2)	Not Null,
	New_Mark	decimal(5,2)	Not Null,
	Changed_On	date		Not Null
);

Create Function TRF_Log_Registration_Mark_Changes()
Returns Trigger
Language plpgsql
As
$body$
Begin
If New.Mark != Old.Mark Then
	Insert Into Registration_Mark_Changes
		(StudentID, CourseID, Semester, Old_Mark, New_Mark, Changed_On)
	Values
		(Old.StudentID, Old.CourseID, Old.Semester, Old.Mark, New.Mark,
	Current_Date);
End If;
Return New;
End;
$body$
```

```
Create Trigger TR_Log_Registration_Mark_Changes
Before Update Of Mark
On Registration
For Each Row
Execute Procedure TRF_Log_Registration_Mark_Changes();
```

Testing:

```
Update          Registration
Set             Mark = 100
Where           StudentID = 199899200 and
                CourseID = 'ANAP1525' and
                Semester = '2000S';

Select StudentID, CourseID, Semester, Old_Mark, New_Mark, Changed_On
From   Registration_Mark_Changes;
```

Create an After Insert trigger on the Payment table to update the student's BalanceOwing is updated when a payment record is inserted. An After trigger ensures that the payment insert is successful before updating the Balance Owing column.

```
Create Function TRF_Payment_Update_Student_BalanceOwing()
Returns Trigger
Language plpgsql
As $body$
Begin
        Update              Student
        Set         BalanceOwing = BalanceOwing - New.Amount
        Where       StudentID = New.StudentID;
        Return New;
End;
$body$;

Create Trigger TR_Payment_Update_Student_BalanceOwing
After Insert
On Payment
For Each Row
Execute Function TRF_Payment_Update_Student_BalanceOwing();
```

Testing:

```
Select StudentID, BalanceOwing
From   Student
WhereStudentID = 200312345;
Insert Into Payment
        (PaymentID, PaymentDate, Amount, PaymentTypeID, StudentID)
Values
        (99, Current_Date, 900, 1, 200312345);
```